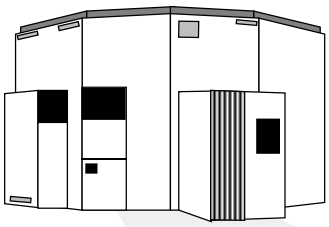
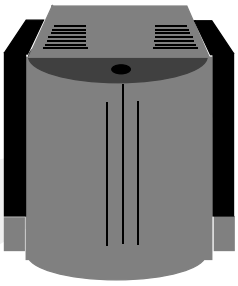
Naval Research Laboratory

Washington, DC 20375-5320





Java Program Development

Course Notes

Instructor: Michael G. Vonk
(202)767-3884
michael.vonk@nrl.navy.mil

Java Program Development

Introduction

This course introduces Java—a portable, object-oriented programming language and supporting run-time environment. Java can be used to create dynamic, interactive Web pages (applets) and standalone applications. Java was created by James Gosling of Sun Microsystems and was formally announced in May 1995.

Topics to be covered in this course include:

- applets, applications, and the Java program development cycle
- object-oriented programming in Java
- language details, data types, and control structures
- graphical user interfaces and the Abstract Windowing Toolkit (AWT)
- input, output, and multimedia
- the Java Application Programming Interface (API)

Some prior programming experience, although in no particular language, is expected as a prerequisite for this class.

Programs developed in this class were tested using version 1.0.2 of the Java API and Sun's Java Development Kit (JDK).

Note: Version 1.1 (or later) of the Java API is now available, and should be used for all new program development.

Class Format

This class is designed to be covered in a series of modules (each with their own lab session) over a period of three days:

	Module
Day 1	1. Applets and Applications
	2. Object-Oriented Programming (Part One)
	3. Language Details
	4. Applet Details
	5. Basic Drawing
Day 2	6. GUI Components
	7. Event Handling
	8. Component Layout
	9. Additional Components
Day 3	10. Object Oriented Programming (Part Two)
	11. Input and Output
	12. Multithreading
	13. Multimedia

A Web page containing all the examples from this class, as well as pointers to additional information is available at:

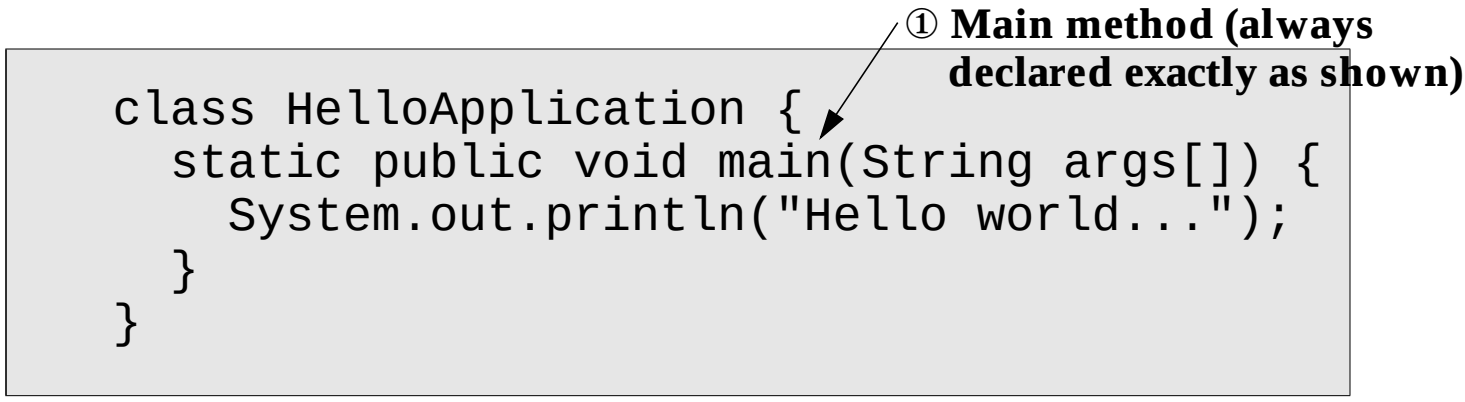
```
http://amp.nrl.navy.mil/
  code5595/ccs-training/java
```

1. Applets and Applications

Java programs can be written as either standalone "*applications*" (executed from the command line) or as "*applets*" that execute in a Web browser.

1.1. Java Applications

The following example is the ubiquitous "hello world" program written as a standalone application:



Example 1 HelloApplication.java

This program consists a "*class*" definition (every Java program has at least one) containing a single "*method*" (`main`) to print a message on the screen.

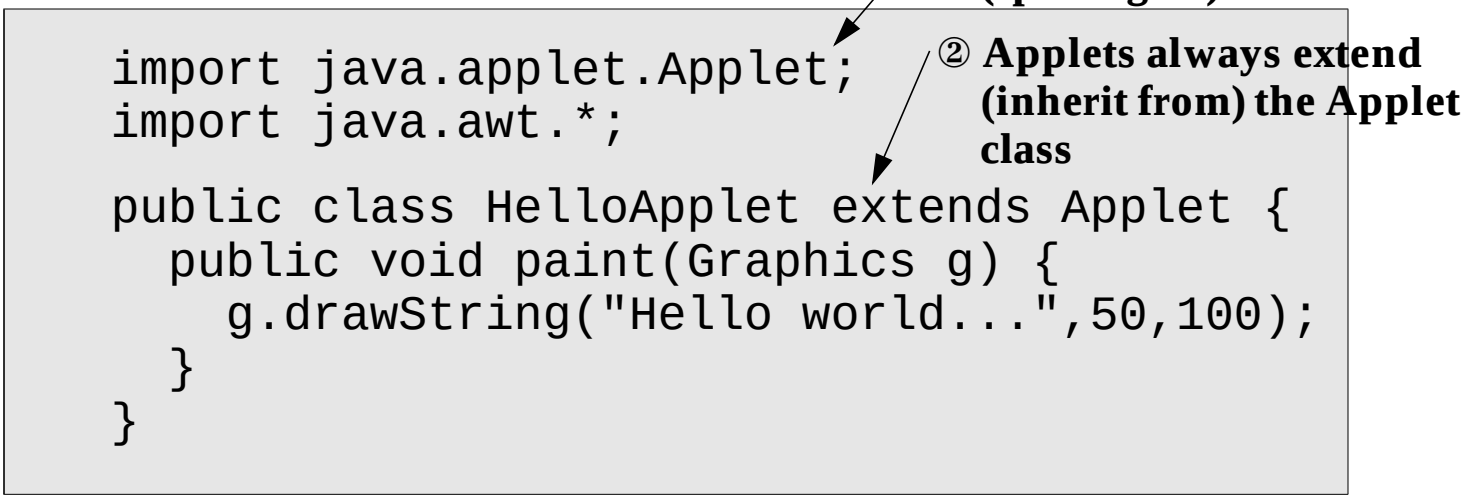
Java source files must be named "*`classname`*`.java`" and are compiled (`javac`) into a platform-independent "*bytecode*" format to create "*class files*." Standalone applications are executed by passing the main class file to the Java interpreter (`java`):

```
% javac HelloApplication.java
% java HelloApplication
Hello world...
%
```

1.2. Java Applets

Java applets can be used to create dynamic, interactive Web pages. There are two steps in this process—first the applet code must be written and compiled; then the class file is referenced from an HTML file.

The following example shows the "hello world" program written as a Java applet:




```
import java.applet.Applet;
import java.awt.*;

public class HelloApplet extends Applet {
  public void paint(Graphics g) {
    g.drawString("Hello world...",50,100);
  }
}
```

Example 2 HelloApplet.java

Java provides a number of predefined classes, one of which is the Applet class, which serves as a template for displaying and interacting with applets via a Web browser. The HelloApplet class above "*extends*" the functionality of the Applet class to paint the hello message into a window on the browser.

Applets are compiled into class files as before:

```
% javac HelloApplet.java
```

The HTML `applet` tag is used to reference an applet's class file, creating an area on the Web page for the applet with a specified width and height:

```
<title>Hello World Applet</title>
Hello world applet:<p>

<applet code   = "HelloApplet.class"
        width  = "150"
        height = "100">
</applet>
```

Example 3 HelloApplet.html

The HTML file containing the applet can be viewed within a browser. Use shift-reload to refresh the page if changes have been made.

The Java Development Kit's (JDK) appletviewer* program can also be used to display an applet:

```
% appletviewer HelloApplet.html
```

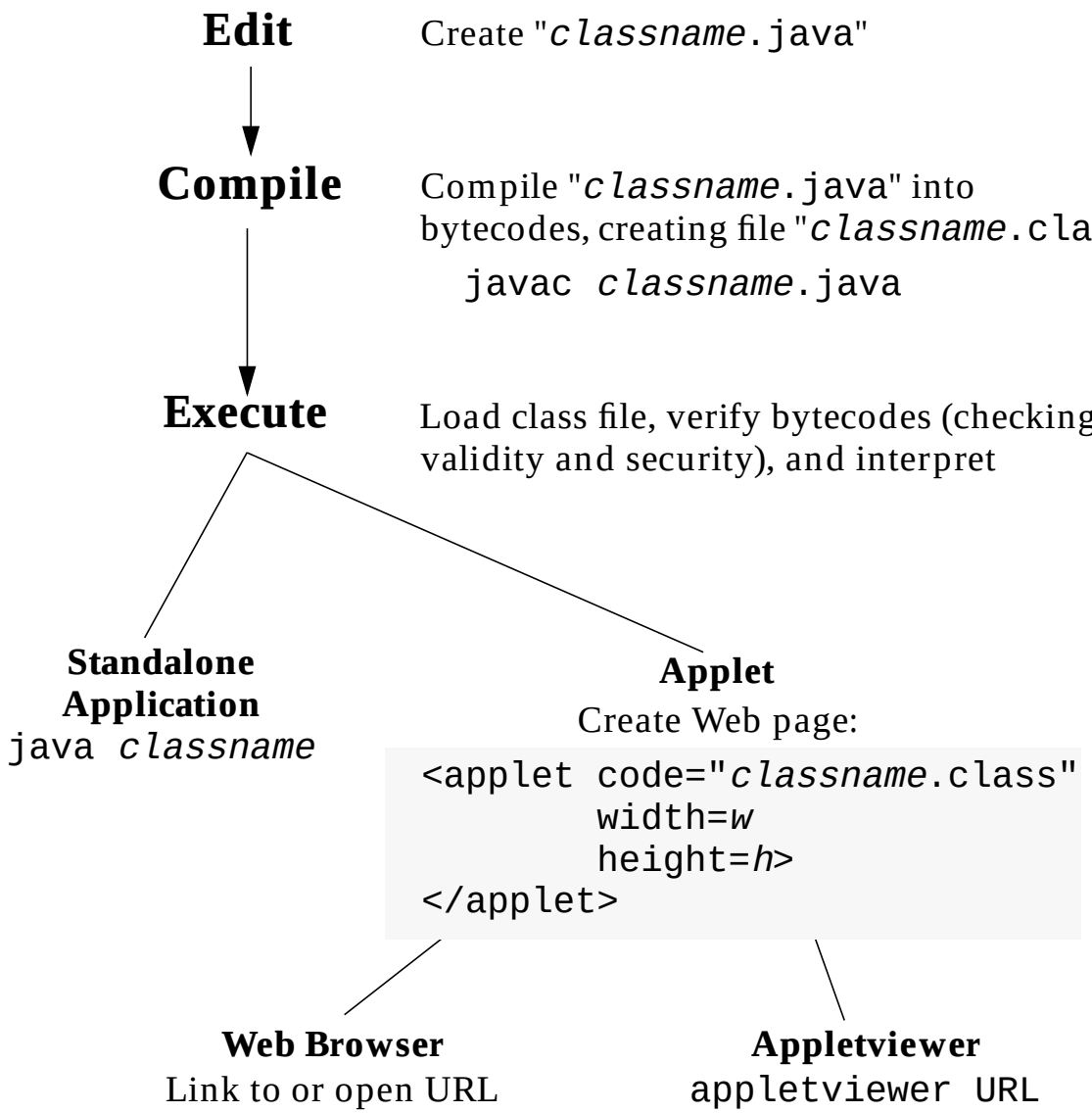
Appletviewer requires an HTML file to load an applet.

* **Note:** Due to problem specific to our X terminal setup, appletviewer does not work properly in the CCS training room.

1.3. Program Development Cycle

Sun's Java Development Kit (JDK) is freely available (see the companion page for reference) and runs on a variety of systems. It includes a compiler, interpreter, and appletviewer.

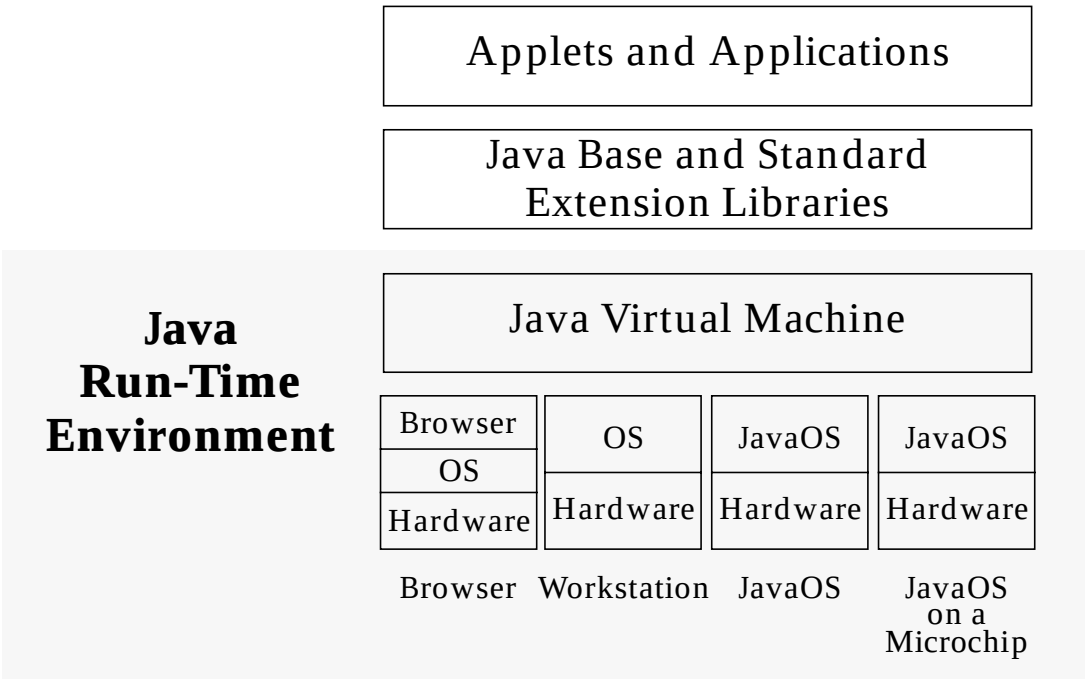
The following diagram summarizes the steps in the Java development cycle:



1.4. Why Java?

Java is an important new programming paradigm for several reasons:

- Platform independence—the Java Virtual Machine (JVM), which translates Java programs into native machine code, provides a standard interface across all platforms



- Simplicity—Java is a streamlined successor to C/C++, with several harmful features removed and additional features added, including standardized class libraries
- Safety—Java applets have several security restrictions (no pointers to access random memory, no access to local files, robust exception handling, bytecode verification, etc.)
- High performance—"just in time" and native machine architecture compilers can be used to achieve speeds rivaling that of C++

2. Object-Oriented Programming (Part One)

In this section, we introduce object-oriented terminology as used in Java and develop some basic examples.

2.1. Terminology

Classes in an object-oriented programming language are similar to user-defined data types in traditional languages. However, in addition to containing data, classes also contain methods for operating on that data.

Classes are used to define a group of objects which share similar attributes and behavior:

Class	Object template—defines a set of variables and methods that operate on those variables
Class Instance	An actual object

Writing a program involves constructing a set of classes. During execution, the actual instances of these classes (the "objects") are created.

Classes are comprised of two components:

Variables	Define the attributes of a particular object or of the class as a whole
Methods	Define the operations that can be performed on an individual object or on all objects in the class

Example (Part 1)

Suppose we want to keep track of the cars we own (assuming we could afford more than one). We might be interested in the following:

- the attributes (make, model, etc.) of each car
- the total number of cars (a class-wide attribute)
- being able to get a description of each car (a method)

These details could be encoded into a class definition as follows:

```
// Car template
class Car {
   String make,
          model;

   static int numCars = 0;

   // Method for returning car description
   public String toString() {
       String description;
       description = make + " " + model;
       return description;
   }
}
```

① "**Instance**" **variables**

② "**Class**" **variable**

③ **Local variable**

Example 4 Car.java

Later on we will write a program that will create car objects, set their attributes, and print their descriptions.

2.2. Variable Scope

Variables define the attributes of an individual object or of a class of objects. Java has three kinds of variables:

Instance Variables	Define attributes of a particular object (eg. the make of a specific car). Each class instance (object) has its own copy of all instance variables.
Class (or "Static") Variables	Represent "class-wide" attributes (eg. the number of cars). Only one copy of a class variable exists.
Local Variables	Store temporary information for use in a method; reinitialized each time the method is called.

Instance variables can be accessed directly (within the class definition itself) or through an object:

```
make
myCar.make
```

Class variables can be accessed directly, through an object of its class, or through the class name:

```
numCars
myCar.numCars
Car.numCars
```

Local variables are only accessible inside the method in which they are defined.

Example (Part 2)

We will now write a program that uses our Car class definition to create a new car, which involves the following steps:

1. Declaring an object
2. Creating ("*instantiating*") the object using the `new` command
3. Setting object and class-wide attributes

Methods related to the object can also be called.

```
class CarTest {
  static public void main(String args[]) {
    // Declare and create a new car
    Car myCar;
    myCar = new Car();

    // Set attributes
    myCar.make = "Mazda";
    myCar.model = "MX-3";
    Car.numCars = Car.numCars + 1;

    // Print car information
    System.out.println("My car is a " +
                       myCar.toString());
    System.out.println("Number of cars: " +
                       Car.numCars);
    }
}
```

① **Refer to instance variables through object**

② **Refer to class variables through class**

Example 5 CarTest.java

2.3. Methods

Methods define the behavior of an object—what happens when an object is created and other operations that can be performed on the object. There are two types of methods:

Instance Methods	Apply to a particular object (eg. to return a description of a specific car). An instance method is simply referred to as a "method."
Class (or "Static") Methods	Apply to the class as a whole (eg. to return the total number of cars).

Method definitions have four* basic parts:

1. Method name
2. Return type (or `void`)
3. Parameter list
4. Body

A method's name, return type, and parameter list are referred to as its "*signature*." Several methods can have the same name, but different signatures. This is known as "*method overloading*."

* **Note:** Methods can also have an access qualifier (which specifies who can execute the method) and a `throws` clause (which defines what errors the method might generate).

2.4. Constructors

A special type of method called a "*constructor*" is used to initialize objects. Constructor methods aren't called directly—they are invoked automatically each time an object of that class is instantiated. Constructors have the same name as the class:

```
Car newCar = new Car();
```

Every class has at least one constructor. If you don't define your own, a Java-supplied default constructor is invoked which takes no arguments and performs no special initialization.

The following constructor could be added to the Car class :

```
public Car(String theMake,String theModel) {
    make = theMake;
    model = theModel;
    numCars = numCars + 1;
}
```

Rather than creating a new car object, setting its attributes, and incrementing the car total in separate steps, our new constructor can be called:

```
Car newCar = new Car("Mazda","Miata");
```

Notes: Unlike regular methods, constructor methods don't have a return type.

Writing your own constructor hides the default (no argument) constructor.

2.4.1. The `this` Keyword

Local variables effectively hide instance variables of the same name. The `this` keyword refers to the current object and can be used to reference the "hidden" instance variable. For example:

```
public Car(String make,String model) {
    this.make = make;
    this.model = model;
    numCars = numCars + 1;
}
```

The `this` keyword can be used only in the body of instance methods.

2.4.2. Class Methods

Class ("static") methods apply to the class as a whole. For example:

```
public static void incrementNumCars() {
    numCars = numCars + 1;
}
```

Static methods can be called without having to instantiate an object. They do not need any data except what they are passed as arguments and whatever static variables are defined in the class. Because static methods do not apply to individual class instances (objects), they have some restrictions:

- Static methods cannot access non-static variables or methods
- There is no `this` keyword in class methods

2.5. Method Overloading

Methods in Java can be overloaded—ie. you can create several methods with the same name, but with different signatures and different definitions. Method overloading is used if a piece of code logically performs the same function, but has a different number or type of arguments.

For example, the Car constructor shown previously could be overloaded:

```
public Car(String make,String model) {
    this.make = make;
    this.model = model;
    numCars = numCars + 1;
}

public Car() {
    numCars = numCars + 1;
}
```

When you call an overloaded method, Java matches up the method name and number, type, and order of arguments to choose which method is executed.

Many system defined methods are "*polymorphic*." For example, the `println()` method can take as an argument a string, an integer, or any of several other data types.

Note: You cannot change only the return type—this results in a compiler error. Variable names in the parameter list are irrelevant.

2.6. Objects Summary

Objects model real-world objects or abstract objects (such as events). There are several steps in the design of a class of objects. You must define their:

1. Attributes

"*instance variables*"	pertain to individual objects
"*class variables*"	pertain to the class as a whole

2. Behavior

"*instance methods*"	operate on individual object
"*class methods*"	operate on the class as a whole

New objects must first be declared and then are "*instantiated*" by calling their class's "*constructor*" method (which performs any necessary initialization).

Where methods logically perform the same functionality on different types of arguments, "*polymorphism*" is used to "*overload*" a method's definition.

Additional aspects object oriented programming will be discussed in Object Oriented Programming (Part Two), including:

- "*encapsulation*" (hiding class details from users)
- "*inheritance*" (defining a new "*subclass*" by extending a previously defined "*superclass*")
- "*packages*" (class libraries)

3. Language Details

When beginning to program in any language, there are several basic things which you must learn—what data types are available, how to declare variables and arrays, how to convert between data types, and how to specify operators and control structures.

3.1. Variables and Data Types

Java is a "*strongly-typed*" language—all variables must be declared to be of a specific type, which can be any of the following:

- a primitive data type
- a class name (ie. the variable is an object)
- a string (a special type of object)
- an array

3.1.1. Variable Declarations

Variable declarations consist of a type and a variable name:

```
int age;
Car myCar;
String name;
```

Variables can be initialized on the declaration statement:

```
int age = 35;
String name = "Michael";
```

Note: Declarations can be intermingled with code.

Constants can be declared by using the keyword `final` and specifying an initial value:

```
final float PI = 3.14;
static final int MAXCARS = 20;
```

Local variables must be given values before they are used or a compiler error will be generated. Instance and class variables do not have this restriction and have default values based on their type:

`null`	for class instances (objects)
`0`	for numeric variables
`'\0'`	for character variables
`false`	for boolean variables

3.1.2. Naming Conventions

Java is "*case sensitive*." Although not required, certain naming conventions make Java programs more readable:

- Constants are in uppercase (`PI`)
- Class names are capitalized (`Car`)
- Variable and method names have initial lowercase letters (`myCar`)

Identifier names can consist of letters, digits, underscores, or dollar signs. They cannot, however, begin with a digit. Names are often made up of several words, with the first word lower-case, and subsequent words capitalized:

```
Button theSelectedButton;
```

3.1.3. Primitive Data Types

The following primitive data types are available:

	type	size (bits)	
Integer	`byte`	8	All integers are signed
	`short`	16	
	`int`	32	
	`long`	64	
Real	`float`	32	IEEE floating point format
	`double`	64	
Character	`char`	16	ISO Unicode character set
Boolean	`boolean`	1	`true` or `false` (booleans are not numbers and cannot be treated as such)

Primitive data types are "*machine independent*"—sizes and characteristics are consistent across all operating systems and architectures.

Notes: Real literals (like `23.79`) are considered double by default (you could use `23.79f` to make it a float)

Character literals are enclosed in apostrophes (strings are enclosed in quotation marks)

3.1.4. Objects

Creating objects is a two-step process—first you declare a variable to be of a particular class and then you use the `new` keyword to instantiate the object:

```
Circle c;
c = new Circle();
```

In the first step, only a reference to a Circle object is created. The second step actually creates an instance of the class. These two steps can be combined in one statement as follows:

```
Circle c = new Circle();
```

3.1.5. Strings

There are two types of string objects in Java—`String` (read only) and `StringBuffer` (modifiable). Strings are instantiated as follows:

```
String hello = "Hello world...";
```

The length of a string can be found using its `length()` method:

```
length = hello.length();
```

3.2. Arrays

To use an array, you first declare the type of data it will contain, and then allocate memory for its contents. Array declaration can be performed using any of the following:

```
datatype array-name[];
datatype[] array-name;
datatype[] array-name[];
```

The brackets can be placed after the variable name, after the type specifier, or both. All are equivalent (although the first method is preferred). For example, to declare an array of `Points`:

```
Point myPoints[];
```

At this point, the only storage allocated is a reference to the array. To allocate storage for the elements of the array, use the `new` keyword and specify the array length:

```
myPoints = new Point[10];
```

The declaration and allocation steps can be combined as follows:

```
Point myPoints[] = new Point[10];
```

3.2.1. Array Initialization

The elements of an uninitiallized array have the following default values:

`0`	for numeric primitive data types
`false`	for boolean variables
`'\0'`	for character arrays
`null`	for everything else

Arrays can be initialized during allocation as follows:

```
String names[] = {"jim", "sue", "john"};
```

Array length is determined from the number of initializers.

3.2.2. Accessing Array Elements

Individual array elements are accessed via their (zero-based) subscripts:

```
names[2] = "steve";
```

Array "*bounds checking*" is performed automatically—attempts to access elements outside array bounds generate run-time exceptions.

3.2.3. Array Length

The number of elements in an array can be found using `length`:

```
numPoints = myPoints.length;
```

Note: There are no parantheses after `length`.

3.2.4. Multi-Dimensional Arrays

Java does not directly support multi-dimensional arrays. They can be faked, however, by allocating arrays of arrays:

```
int coords[][] = new int[12][12];
coords[1][2] = 1;

int b[][] = { {1,2,3},
              {2,3,4} };
```

3.3. Type Conversion

Java's "type wrapper" classes provides a means of converting strings to numbers:

```
int i = Integer.valueOf("406").intValue();
float f = Float.valueOf("3.1").floatValue();
double d =
   Double.valueOf("1.23").doubleValue();
```

Numbers can be converted to strings using:

```
String.valueOf(myNumber);
```

Each type in a mixed-type numeric expression is promoted to the "highest" type in the expression. Converting to a "lower" type can lead to loss of precision and incorrect results. To do so, you must explicitly call a "cast" operator:

```
intResult = (int) myFloat / myInt;
```

Objects can be converted to strings using either the Object class's or your own `toString()` method.

3.4. Argument Passing

Passing arguments to a method occurs in one of two ways:

- Primitive types are passed "*by value*" (a copy of the argument is passed—you cannot change the original)
- Objects are passed "*by reference*" (a pointer to the original data is passed—changes made within the method affect data in the calling method)

Arrays are treated as objects. Individual array elements (if of a primitive data type), however, are passed by value.

3.5. Java Syntax

Several other Java language details should be noted:

- all statements end in semicolons
- compound statements, or blocks, can be placed wherever a single statement can be placed and are surrounded by `{}`; variables can be declared for use within blocks

Three types of comment delimiters can be used:

`// comment`	single line comments (continue until end of line)
`/* comment */`	multi-line comments (can't be nested)
`/** comment */`	special comments used with `javadoc`

3.6. Operators

The following operators are available in Java:

String	`+` concatenation (ints ⇒ strings)

Numeric	`+`
	`-`
	`*`
	`/` int / int ⇒ int, otherwise real
	`%` modulus (yields integer)

Comparison*	`==` `!=`
	`<` `<=`
	`>` `>=`

Boolean	`&&` `||` and / or (short circuit) `&` `|` and / or (non-short circuit) `^` exclusive or `!` not
Assignment	`a` *`op`*`= b;` shorthand for `a = a` *`op`* `b;`
Autoincrement (decrement)	`a++` `a--` `++a` `--a`
Cast	(*`datatype`*) *`variable`*
Conditional	*`test`*`? `*`a`*`:`*`b`*

Refer to a precedence table for order of evaluation (or use parantheses to explicitly specify order).

* **Note:** Comparison operators cannot be used with objects or strings—use *`object1`*`.equals(`*`object2`*`)`

3.7. Control Structures

Java's conditional and looping constructs are similar to those used in C:

Condition	`if (`*`condition1`*`) {`
	`    `*`statement-block`*
	`}`
	`else if (`*`condition2`*`) {`
	`    `*`statement-block`*
	`}`
	`else {`
	`    `*`statement-block`*
	`}`

Switch	`switch (`*`test`*`) {`
	`    case `*`one`*`:  `*`statement-list`*`;`
	`                 break;`
	`    case `*`two`*`:  `*`statement-list`*`;`
	`                 break;`
	`       .`
	`       .`
	`       .`
	`    default:   `*`statement-list`*`;`
	`                 break;`
	`}`

Looping	`for (`*`init`*`;`*`condition`*`;`*`increment`*`) {`
	`    `*`statement-block`*
	`}`
	
	`while (`*`condition`*`) {`
	`    `*`statement-block`*
	`}`

Several points about these control structures must be noted:

- The *`condition`* evaluated must be a boolean value.
- The switch *`test`* must be a simple primitive type that is castable to `int` (you cannot use long or float or objects).
- Single statements can be used instead of the bracket-enclosed statement blocks shown above. This can, however, lead to the "*dangling-else*" problem.
- The following two statements can be used for loop control:

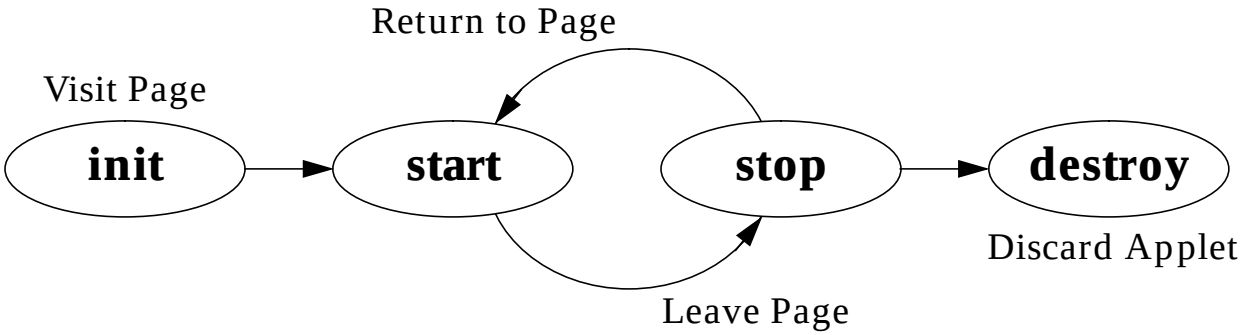
`continue`	jump to next iteration of loop
`break`	terminate loop

4. Applet Details

In this section we discuss several aspects specifically related to Java applets—their life cycle, context, customization, and security restrictions.

4.1. Applet Life Cycle

Applets don't have a programmer-supplied main method. The main method is defined in the browser and provides a means of interacting with the user. The following diagram shows the four methods called in the life cycle of an applet:



These four "milestone" methods are used as follows:

- init() Used to initialize the applet (create GUI components, load sounds / images, and create threads)
- start() Called when the page is first visited and every time the browser returns to the page (start or restart animations and threads, etc.)
- stop() Called when the page is left (stop animations and threads)
- destroy() Called when the applet is garbage collected (release any other resources)

Other frequently called methods:

- paint(Graphics g)
  Called after `init()` is finished and start has begun executing to draw (paint) the applet. Called automatically every time applet needs to be redrawn.
- repaint() Schedules call to component's update method. Can be used to call the paint method explicitly to redraw the applet.
- update() Responsible for redrawing applet, default version redraws background and then calls paint.

Applet always begin with a series of three method calls—init, start, and paint. All of the methods discussed above do nothing if not redefined (overridden) by the programmer.

4.2. Applet Context and Status

The following two methods return information about an applet:

- getCodeBase() Returns URL of directory containing the applet's class file
- getDocumentBase() Returns URL of directory containing the Web page referencing the applet

Applets can display messages in the browser's status bar using:

```
showStatus("Applet starting...");
```

4.3. Passing Arguments to Applets

Applets can be customized by specifying a list of arguments to be passed to the applet via the HTML `param` tag:

```
<applet code="xxx.class" width=w height=h>
<param name="user" value="John Doe">
</applet>
```

The `param` tag must appear between applet tags. Each parameter has a name and value. The `getParameter()` method is used to get a parameter's value:

```
String theValue = getParameter("user");
```

4.4. Applet Security Restrictions

Applets have certain restrictions (that standalone applications don't have) that help prevent them from causing damage to the system or security breaches. These restrictions include:

- applets can't read or write to the local file system
- applets can't communicate with a server other than the one from which it came
- applets can't run programs on or load libraries from the local system
- applets can't read certain system properties
- applet windows are marked as "untrusted"

5. Basic Drawing

The AWT Graphics package provides methods to draw text, shapes, and images. To do so, you must understand what is known as the "*graphics context*," Java's coordinate system, the various drawing methods, and how to select fonts and colors.

5.1. Graphics Context

Every user interface component (including the applet window) has an associated graphics context, which defines:

- component on which to draw
- translation origin
- clipping rectangle
- current color
- current font
- current paint mode (replace or XOR)
- current XOR alternation color

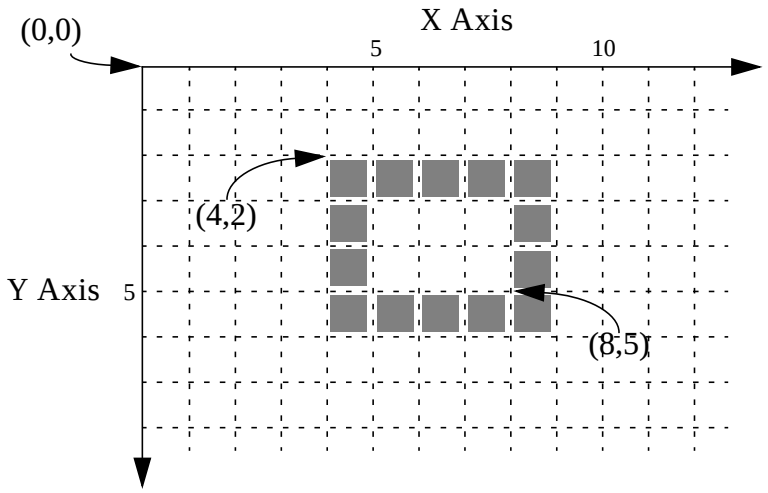
In the Hello World applet, the paint method takes one argument, the graphics context associated with the applet window:

```
public void paint(Graphics g) {
    g.drawString("Hello world...",50,100);
}
```

The graphics context (object) will be named "`g`" in the examples in this section.

5.2. Coordinate System

Java's coordinate system starts in the upper left-hand corner of the component with point (0,0) and is measured in pixels. X values increase to the right and y values increase downward:



In the figure above, a rectangle is drawn with the upper-left corner at coordinate (4,2) and a width and height of 4 and 3, respectively. The method used to draw this rectangle would be:

```
g.drawRect(4,2,4,3);
```

Note: Drawing occurs below and to the right of the specified point—the rectangle above appears one pixel wider and taller than you might expect. This applies to other outlined shapes as well. Filled shapes are drawn inside the specified rectangle, making them one row shorter and skinnier than their outlined counterparts.

5.3. Drawing Methods

Basic drawing methods include:

Text	`g.drawString(`*`string`*`,`*`x`*`,`*`y`*`)`
Lines	`g.drawLine(`*`x1`*`,`*`y1`*`,`*`x2`*`,`*`y2`*`)`

Rectangles	`g.drawRect(`*`x`*`,`*`y`*`,`*`width`*`,`*`height`*`)`
	`g.fillRect(`*`x`*`,`*`y`*`,`*`width`*`,`*`height`*`)`
	`g.clearRect(`*`x`*`,`*`y`*`,`*`width`*`,`*`height`*`)`

Rounded Rectangles	`g.drawRoundRect(`*`x`*`,`*`y`*`,`*`width`*`,`
	`  `*`height`*`,`*`arcWidth`*`,`*`arcHeight`*`)`
	`g.fillRoundRect(`*`x`*`,`*`y`*`,`*`width`*`,`
	`  `*`height`*`,`*`arcWidth`*`,`*`arcHeight`*`)`

3D Rectangles	`g.draw3DRect(`*`x`*`,`*`y`*`,`*`width`*`,`*`height`*`,`
	`  `*`boolean`*`)`
	`g.fill3DRect(`*`x`*`,`*`y`*`,`*`width`*`,`*`height`*`,`
	`  `*`boolean`*`)`

Ovals	`g.drawOval(`*`x`*`,`*`y`*`,`*`width`*`,`*`height`*`)` `g.fillOval(`*`x`*`,`*`y`*`,`*`width`*`,`*`height`*`)`

Arcs	`g.drawArc(`*`x`*`,`*`y`*`,`*`width`*`,`*`height`*`,`
	`  `*`startAngle`*`,`*`arcAngle`*`)`
	`g.fillArc(`*`x`*`,`*`y`*`,`*`width`*`,`*`height`*`,`
	`  `*`startAngle`*`,`*`arcAngle`*`)`

Polygons	`g.drawPolygon(`*`x`*`[],`*`y`*`[],`*`n`*`)` `g.fillPolygon(`*`x`*`[],`*`y`*`[],`*`n`*`)`

Several points should be noted when using the basic drawing methods:

- Strings are drawn above and to the right of the specified point.
- Outlined shapes are one pixel taller and wider than their corresponding filled shapes.
- 3D rectangles have an additional argument which is used to specify a raised rectangle (if true) or lowered rectangle (if false). Shadows are highlighted in a color determined by Java, and may not show up if drawing in black on a white background, or vice versa.
- The upper-left corner of rectangles, ovals, and arcs is specified along with a width and height (both of which must be positive). The following could be used to draw a rectangle, given any two points:

```
g.drawRect(Math.min(x1,x2),
           Math.min(y1,y2),
           Math.abs(x1-x2),
           Math.abs(y1-y2));
```

- The AWT does not currently support line widths (can simulate by drawing multiple lines at one pixel offsets or by drawing filled rectangles).
- The AWT also does not support fill or line patterns.

5.4. Fonts

Any font available on the local system can be used to draw text. A new Font object must first be created, and then the font associated with the current graphics context can be set:

```
Font myFont;
myFont = new Font(fontName,style,size);
g.setFont(myFont);
```

Commonly available fonts include "Courier", "Helvetica", and "Times Roman". Three styles are available:

- Font.PLAIN
- Font.BOLD
- Font.ITALIC

Font size is measured in points (1 point = 1/72 inch).

A list of available font names can be obtained using the current Toolkit object's `getFontList()` method:

```
Toolkit tk = getToolkit();
String fontList[] = tk.getFontList();
```

5.5. Colors

Two colors affect drawing—a component's background color and the current drawing color, both of which can be changed easily using Color objects. RGB color values can be specified using either integer for floating point numbers:

```
Color myColor = new Color(0,0,255);
Color newColor = new Color(0.12,0.76,0.05));
```

Integer values are in the range 0-255. Floating point values in the range 0.0-1.0 may also be specified, but are internally converted to integers. A set of 13 predefined color constants is also available, including:

```
Color.blue
Color.orange
```

The drawing and background colors can be set as follows:

```
g.setColor(myColor);
setBackground(Color.lightGray);
```

The `getColor()` method can be used to obtain the current drawing color. Integer values of a color's RGB triplet can also be obtained:

```
currentColor = g.getColor();
redValue   = currentColor.getRed();
greenValue = currentColor.getGreen();
blueValue  = currentColor.getBlue();
```

Note: Component background color is set using the component's methods, not the graphics object.

6. Graphical User Interface Components

A Graphical User Interface (GUI) is used to control interaction between the user and a Java applet or window-based application. GUIs are built from components (also called "widgets") including push buttons, text fields, lists, and menus.

The following sequence of steps are used with components:

- declaring and creating components
- assigning component attributes
- adding components to a container
- writing event handling code

For components to be visible, they must be added to a container (such as an applet or window). A default (or user-specified) layout manager arranges components within containers. Events, generated when components are selected, are handled by event handling code.

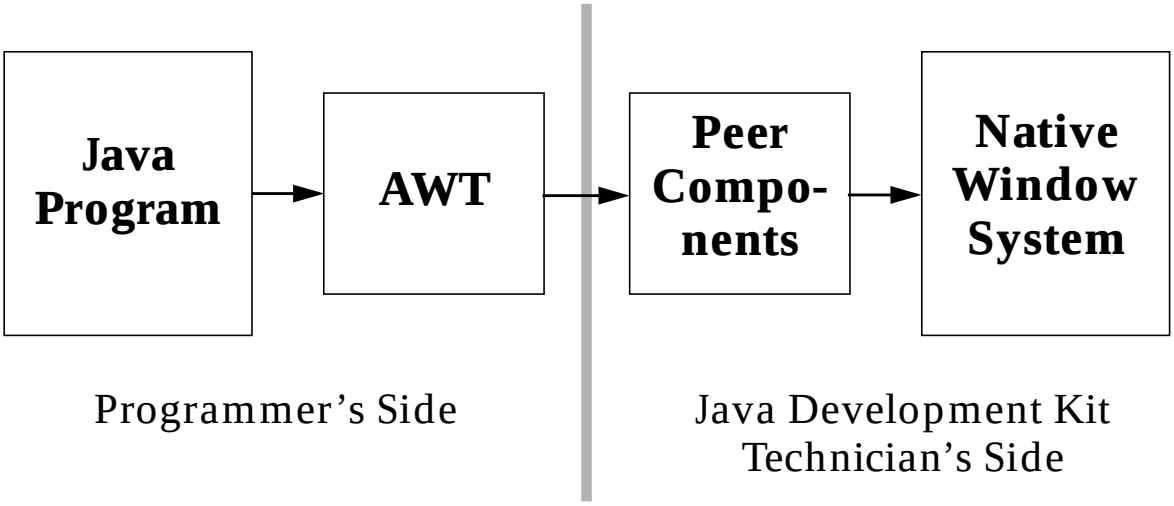
The following sections describe the various components, layout managers, and event handling methods.

6.1. Abstract Windowing Toolkit

The Abstract Windowing Toolkit (AWT) provides a generalized set of classes for building graphical user interfaces and can be used without concern for platform-specific windowing issues:

- Uses the native window system, i.e. Motif, MS Windows
- Subtlely different behavior between window systems can produce subtlely different behavior for your application from system to system
- Java supports “highest common factor” (or “lowest common denominator”) of all supported native window toolkits

Platform independence is made possible through the use of AWT classes known as "peers" (native GUI components which are manipulated by the AWT classes). The AWT delegates the actual rendering and behavior of components to these peers.

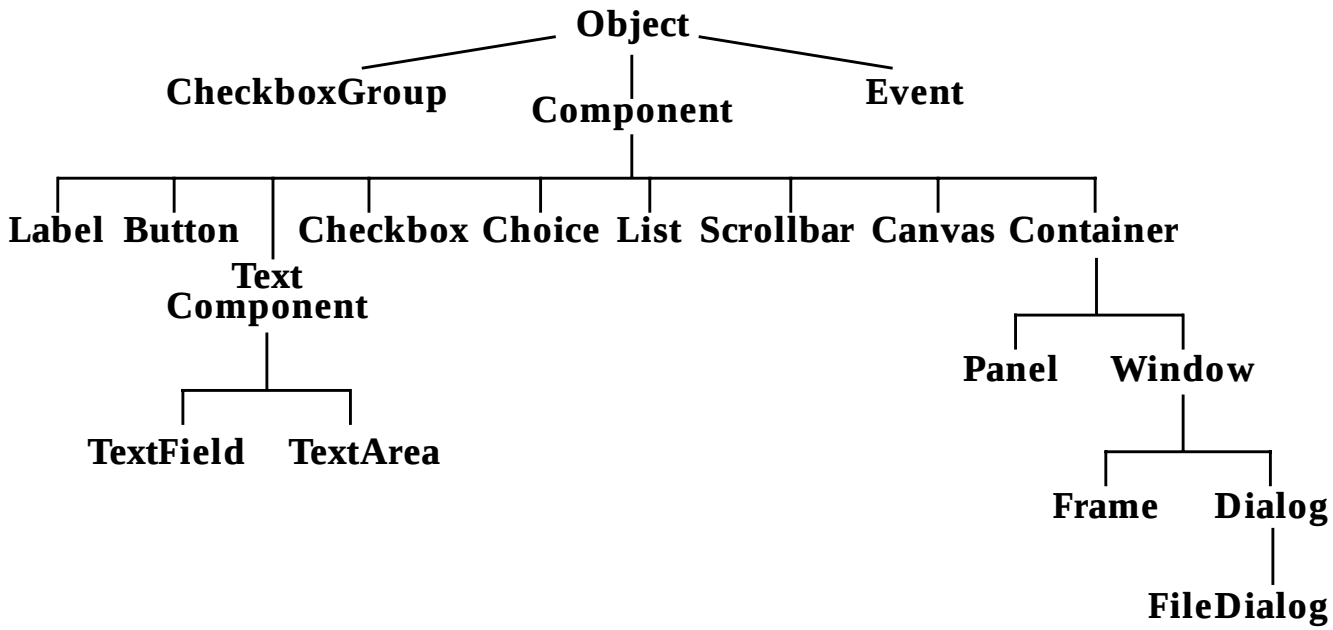


Use of peers enables applets and applications to retain "look and feel" of the native windowing system.

The AWT is comprised of four major parts:

- A set of components
- A set of containers—objects that hold components
- A set of predefined layout managers used in positioning component within containers
- A set of simple graphics operations (discussed in the previous module)

The following diagram illustrates the AWT class hierarchy as it relates to the following modules:



Note: Applets are a special type of panel, and thus can contain components.

6.2. Labels

Labels are single-line, readonly text fields.

Constructors	`Label()` `Label(String s)`
Public Methods	`String theText = getText()` `setText(String s)`
Events	none

The following declares and creates a new Label, and adds it to the interface:

```
Label myLabel;
myLabel = new Label("This is a Label");
add(myLabel);
```

This could be shortened to:

```
Label myLabel = new Label("This is a Label");
add(myLabel);
```

Normally you declare GUI components global with respect to the class so they can be referenced in event handling code. However, since labels do not generate events, the following could be used:

```
add(new Label("This is a Label"));
```

6.3. Push Buttons

Push buttons are labeled rectangles which generate events when clicked with the mouse.

Constructors	`Button()` `Button(String s)`
Public Methods	`String theLabel = getLabel()` `setLabel(String s)`
Events	`action` event generated when user clicks on button

The following declares, creates, and adds a button:

```
Button myButton;
myButton = new Button("My Button");
add(myButton);
```

Note: Only the most commonly used constructors and public methods are listed in the tables in these notes. Consult the Java API documentation for a complete list.

6.4. Example

The following is a fully functional applet which displays a button and handles button selections:

```
import java.applet.Applet;
import java.awt.*;

public class ButtonApplet extends Applet {
    // Declare components as global to class
    Button   myButton;
    TextArea events;

    // Create user interface
    public void init() {
        // Initialize and add button to layout
        myButton = new Button("My Button");
        add(myButton);

        // Create text area to display event
        events = new TextArea(5,30);
        add(events);
    }

    // Handle component action events
    public boolean action(Event e,Object o) {
        if (e.target == myButton) {
           events.appendText("My button pressed...\n");
            return true;
        }
        else {
            return false;
        }
    }
}
```

Example 6 ButtonApplet

The ButtonApplet example above illustrates the general sequence of steps followed when adding GUI components to the interface:

1. Components are declared global to the class so they can be referenced in all methods of the class (specifically the event handling method).
2. Components are instantiated and added to the interface in the `init()` method, which is called once when the applet is initialized.
3. Component events are handled either in an `action()` method or in a `handleEvent()` method.

Components are automatically painted—you do not have to draw them in a `paint()` method. Event handling is described in detail in the following module.

Note: *DO NOT* instantiate and add components in the `paint()` method, as it is called numerous times during an applet's life cycle.

6.5. Text Fields

Text fields allow users to enter a single line of text. Pressing "Enter" or "Return" in a text field generates an event.

Constructors	`TextField()`
	`TextField(int cols)`
	`TextField(String s)`
	`TextField(String s,`
	`int cols)`

Public Methods	`setEditable(boolean)`
	`setEchoCharacter(char c)`
	`setText(String s);`
	`String myText = getText()`
	`String myText = getSelectedText()`

Events	`action` event generated when user presses Enter or Return in field

The following declares, creates, and adds a text field:

```
// In global declaration section
TextField myTextField;

// In init() method
myTextField = new TextField(20);
add(myTextField);
```

Note: Even though a TextField can be specified as readonly (using `setEditable(false)`), it still generates events.

6.6. Text Areas

Multi-line text input area with vertical and horizontal scrollbars (users can enter as many lines of text as they want).

Constructors	`TextArea()`
	`TextArea(int rows,`
	`int cols)`
	`TextArea(String s)`
	`TextArea(String s,`
	`int rows,`
	`int cols)`

Public Methods	`setEditable(boolean)`
	`setText(String s);`
	`String myText = getText()`
	`String myText = getSelectedText()`

Events	none

The following declares, creates, and adds a four row by 40 column text area:

```
TextArea comments;

comments = new TextArea(4,40);
add(comments);
```

TextAreas do not generate events when the return key is pressed (thus allowing newlines to be entered by user). Retrieving text from a TextArea requires an external event, such as a button press.

6.7. Checkboxes

Checkboxes are used to get boolean values from the user.

Constructors	`Checkbox(String label)`
	`Checkbox(String label,`
	`CheckboxGroup cbg,`
	`boolean selected)`

Public Methods	`setState(boolean);`
	`boolean state = getState();`
	`String myLabel = getLabel();`

Events	`action` event generated when user checks a box (the state of a box can also be probed)

The following declares, creates, and adds two checkboxes:

```
Checkbox fries,
         coke;

fries = new Checkbox("Fries");
coke = new Checkbox("Coke");
add(fries);
add(coke);
```

Any number of (ungrouped) checkboxes can be checked by the user. CheckboxGroups (see next page) can be used to allow only one selection from a group of checkboxes.

6.8. Checkbox Groups

A set of checkboxes can be grouped together to function as "radio buttons." No more than one checkbox in a checkbox group can be selected at a time.

Constructors	`CheckboxGroup()`
Public Methods	`Checkbox cb = getCurrent()` `setCurrent(Checkbox cb)`
Events	none (events are generated by the individual checkboxes)

The following declares, creates, and adds a group of two checkboxes:

```
CheckboxGroup cbg;
Checkbox mc,
         visa;

cbg = new CheckboxGroup();
mc = new Checkbox("MasterCard",cbg,false);
visa = new Checkbox("Visa",cbg,false);
add(mc);
add(visa);
```

The Checkboxes above take three arguments—a label, a checkbox group object, and a boolean value indicating whether the checkbox is selected by default or not.

6.9. Choice Buttons

A choice button (a.k.a. "popup menu") allows the user to select a single item from a list:

Constructors	`Choice()`

Public Methods	`addItem(String s)`
	`int numItems = countItems()`
	`String item = getItem(int index)`
	`String item = getSelectedItem()`
	`int index = getSelectedIndex()`

Events	`action` event generated when user releases mouse button over choice item

Choice buttons are used by first declaring and creating a Choice object, adding items to it, and finally adding the Choice object to the interface:

```
Choice cars;

cars = new Choice();
cars.addItem("Mazda");
cars.addItem("Volkswagen");
cars.addItem("BMW");
add(cars);
```

Only one item (the first by default) is displayed at a time. When the user presses the popup menu's down arrow, the full list appears from which the user may select.

6.10. Lists

Lists are used to display a long list of items, one per line, along with a scroll bar. By default, only one selection can be made:

Constructors	`List(int numVisibleLines)`
	`List(int numVisibleLines,`
	`boolean multipleSelections)`

Public Methods	`addItem(String s)`
	`String item = getItem(int index)`
	`String items[] =`
	`getSelectedItems()`

Events	`action` event generated when user double-clicks on an item; list select or deselect event generated if user single-clicks on an item

The following declares, creates, and adds items to a list and then adds the list to the interface:

```
List condiments;

condiments = new List(3,true);
condiments.addItem("ketchup");
condiments.addItem("mustard");
condiments.addItem("relish");
add(condiments);
```

Notes: Extraneous list select and deselect events occur when an item is double-clicked.

An external event is generally used to trigger an action with lists that allow multiple selections.

6.11. Scrollbars

Scrollbars allow users to easily select from a range of values using a "slider":

Constructors	`Scrollbar(orientation*,`
	`int initial-value,`
	`int block-increment,`
	`int minimum-value,`
	`int maximum-value)`

Public Methods	`int value = getValue()`
Events	a scrollbar adjustment event is generated if user changes the value of the scrollbar

* `Scrollbar.VERTICAL` or `Scrollbar.HORIZONTAL`

Three types of scrollbar selections can be made. Users can:

1. Click on the left (right) arrow to decrease (increase) the scrollbar value by one
2. Click in the area to the left (right) of the slider to decrease (increase) the scrollbar value by one block increment
3. Drag the slider left or right

The following declares, creates, and adds a horizontal scrollbar:

```
Scrollbar sb;

sb = new Scrollbar(Scrollbar.HORIZONTAL,
                   0,100,0,1000);
add(sb);
```

6.12. Canvases

Canvases are normally used to provide a drawing area within an applet. Rather than creating a new canvas component, the canvas class is usually subclassed:

```
class CanvasSubclass extends Canvas {
    .
    .
    .
}
```

Three methods of the Canvas class are typically overridden:

- `preferredSize()`
- `minimumSize()`
- `paint()`

The first two of these methods are used to size the canvas. The paint method is where all drawing occurs:

```
class CanvasSubclass extends Canvas {
    public void paint(Graphics g) {
        g.drawString("Hello",10,25);
    }
    public Dimension minimumSize() {
        return new Dimension(50,50);
    }
    public Dimension preferredSize() {
        return minimumSize();
    }
}
```

7. Event Handling

An "event loop" is the basis of any Java applet or window-based standalone application. Programs draw their components on the screen and wait for and respond to events that occur within these components. For example, users may click the mouse on a button, select from a list, or drag the mouse in a canvas.

7.1. Event Objects

Event objects are used by event handling methods and have the following fields:

`id`	type of event (there are currently 27 events types), such as `KEY_PRESS`
`target`	component for which the event is intended
`when`	time event occurred
`x,y`	location of cursor when event occurred
`key`	for keyboard events, identifies the key that was pressed
`modifiers`	state of control, meta, and shift keys
`clickCount`	for mouse down events, specifies number of consecutive clicks
`arg`	Event specific information, such as a selected button's label
`evt`	pointer to next event in queue (used by system)

7.2. Event Handling Methods

When an event occurs inside a component, a method of that component is invoked. Their are two ways to respond to an event. You can either:

- Override the `handleEvent()` method
- Override one of the Java-supplied "convenience" methods

You can also choose not to handle an event by either passing it on to the component's container or by simply ignoring it.

7.2.1. Overriding `handleEvent()`

The `handleEvent()` method can be used to handle all types of events. You can perform actions based on the event's type (`id`), the component for which the event was intended (`target`), or the type of component (`instanceof`).

For example:

```
public boolean handleEvent(Event e) {
  switch (e.id) {
    case Event.KEY_PRESS:
        // Handle key press event
        ...
    case Event.KEY_RELEASE:
        // Handle key release event
        ...
    etc.
  }
}
```

You only need to check those events in which you are interested.

7.2.2. Using Convenience Methods

Events of all types can be handled using `handleEvent()`. Java supplies several additional "convenience" methods that can also be used to handle specific types of events:

```
boolean action(Event,Object)

boolean mouseUp(Event,int,int)
boolean mouseDown(Event,int,int)
boolean mouseDrag(Event,int,int)
boolean mouseMove(Event,int,int)
boolean mouseEnter(Event,int,int)
boolean mouseExit(Event, int, int)

boolean keyUp(Event,int)
boolean keyDown(Event, int)

boolean gotFocus(Event, Object)
Boolean lostFocus(Event, Object)
```

The following GUI components generate "action" events:

- Button
- Checkbox
- TextField
- Choice

Events generated within these components can be handled via the `Event.ACTION_EVENT` case in `handleEvent()`, or via the action convenience method:

```
public boolean action(Event e,Object o) {
  // Handle component action event
  ...
}
```

The action method takes an additional Object argument which differs depending on what type of component is targetted:

If the event is from an instance of:	**Then the object is set to a:**
Button	*String* with the text of label of the button.
Checkbox	*Boolean* value—*true* if checkbox is selected and *false* if it isn't.
TextField	*String* value of text field.
Choice	*String* value of selected item.
List	*String* value of selected (deselected) item

7.2.3. Event Propagation

Both `handleEvent()` and `action()` require you to return a boolean value to indicate if the event has been handled or not. You may return:

`true`	indicates event has been completely handled
`false`	propagates event to its container
`super.handleEvent(e)`	lets superclass handle event and decide whether to propagate

Notes: Return `true` if you have completely handled the event, otherwise:

- Return `false` in convenience methods
- Return `super.handleEvent(e)` if you are overriding `handleEvent()`

8. Laying Out Components

In the previous modules, we learned how to create GUI components, add them to the interface, and handle the events they generate. In this module, we learn to how to control the arrangement of components within the interface.

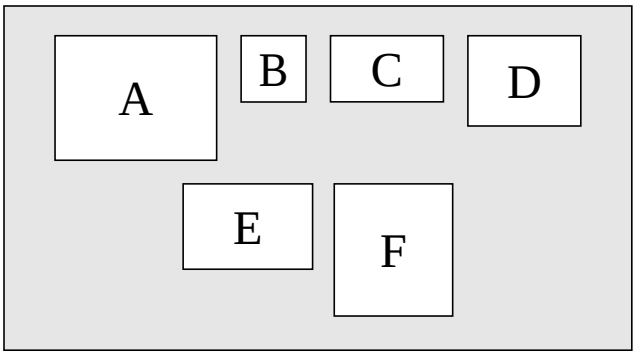
The AWT provides five "layout managers," each with their own rules for positioning and sizing components:

FlowLayout	Adds components left to right, top to bottom (default layout manager for applets and panels)
BorderLayout	Adds components to North, South, East, and West borders and Center of container (default for frames and dialogs)
GridLayout	Adds components to grid of equal-sized cells
CardLayout	Breaks down interface into a deck of "cards," only one of which is visible at a time (each card has its own layout manager)
GridBagLayout	Similar to GridLayout, except cells do not need to be the same size (the most precise, and complicated, of the available layouts)

For all but the most trivial cases, these layout managers are, by themselves, inadequate. However, by grouping components into panels, each taking advantage of the characteristics of their own layout manager, attractive interfaces can be created.

8.1. FlowLayout

With FlowLayout (the default layout manager for applets and panels), components are added from left to right, starting new rows as necessary:



The FlowLayout layout manager can be set for a container (such as an applet) in the `init()` method. Components are then added, as follows:

```
setLayout(new FlowLayout());
add(componentA);
add(componentB);
   .
   .
   .
```

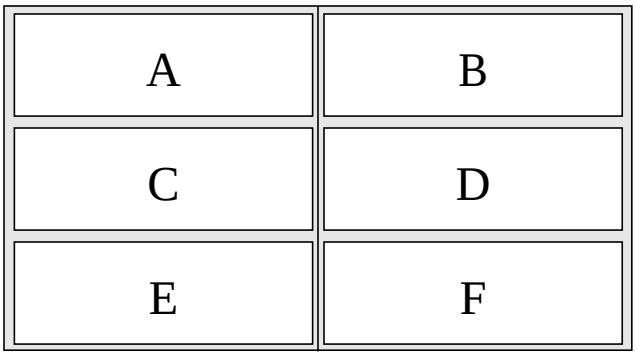
By default, components are centered within each row. Left or right justification can be specified when the FlowLayout object is created. For example:

```
setLayout(new FlowLayout(FlowLayout.LEFT));
```

Note: Each component gets only as much space as it needs.

8.2. GridLayout

With GridLayout, you specify a number of rows and columns. The container is then broken up into a table (grid) of equal-sized cells:



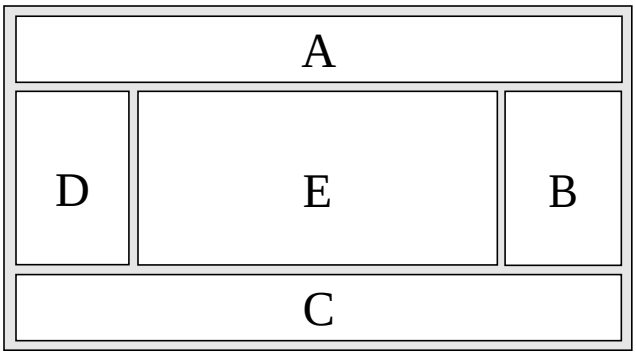
The following can be used to specify a three-row, two-column grid of components:

```
setLayout(new GridLayout(3,2));
add(componentA);
add(componentB);
   .
   .
   .
```

Note: The grid fills the entire container. Components are stretched (or shrunk) to fill the space within their cells.

8.3. BorderLayout

BorderLayout (the default for frames and dialogs) is used to add components to the North, East, South, and West sides of a container, with another component in the Center:



The following can be used to specify BorderLayout and add components:

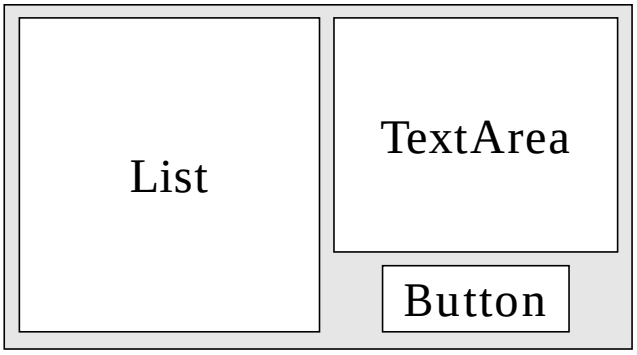
```
setLayout(new BorderLayout());
add("North",componentA);
add("East",componentB);
   .
   .
   .
```

Notes: Components are resized to fill the entire container.

The North and South components are stretched horizontally from the left edge to the right. The East and West components are stretched vertically from the bottom of the North component to the top of the South. The Center component expands to fill the remaining space.

8.4. Grouping Components in Panels

The layout managers discussed so far are good for laying out small sets of components. To create more complex layouts, components can be grouped together into "panels," with each panel using its own layout manager.



The above interface cannot be created using a single layout manager. We must group the two right components into a panel and then add the list and right panel to the interface:

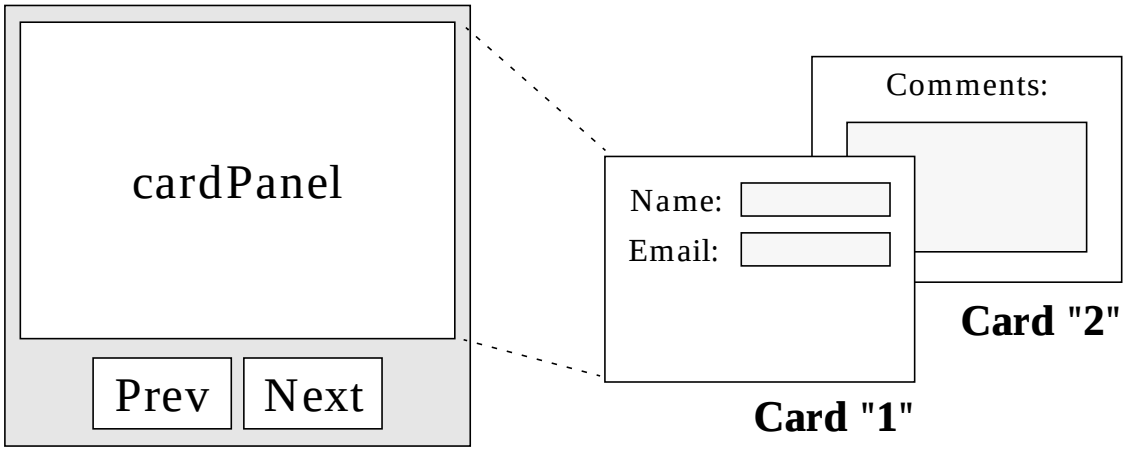
```
// Create panel and set its layout manager
Panel rightPanel = new Panel();
rightPanel.setLayout(new BorderLayout());

// Add text area and button to panel
rightPanel.add("Center",myText);
rightPanel.add("South",myButton);

// Set applet layout and add list and panel
setLayout(new GridLayout(1,2));
add(myList);
add(rightPanel);
```

8.5. CardLayout

CardLayout is analogous to a deck of playing cards:

- You can only see one "card" at a time
- Shuffling to a new "card" requires some event to happen
- Each "card" is typically a panel with its own layout manager

The following example contains two panels. The top panel is layed out using CardLayout and contains two "cards," only one of which is visible at a time. The bottom panel contains two buttons to control which card is displayed:



The following sequence of steps could be used to create this interface:

- Create panels for Card "1" and Card "2"
- Create cardPanel using CardLayout and add card panels
- Create controlPanel containing previous and next buttons
- Add cardPanel and controlPanel to interface

The cardPanel is created as follows:

```
cardPanel = new Panel();
cardPanel.setLayout(cl);
cardPanel.add("1",card1);
cardPanel.add("2",card2);
```

Each card is added to cardPanel using a String identifier (normally a sequence of numbers as shown above). Because the CardLayout object is referenced in the event handler, it is declared global to the class:

```
CardLayout cl = new CardLayout();
```

The following event handler code processes previous and next button presses and sequences through the cards to be displayed in cardPanel:

```
// Handle previous and next button events
public boolean action(Event e, Object o) {
   if (e.target == previous) {
      cl.previous(cardPanel);
      return true;
   }
   else if (e.target == next) {
      cl.next(cardPanel);
      return true;
   }
   return false;
}
```

8.6. Component Sizing

Depending on the layout manager used, components may be resized horizontally and / or vertically to fill the container. The following table shows, for each layout manager, whether a component's preferred size is either respected or ignored:

FlowLayout	Width and height preferences respected (a component's preferred size when using FlowLayout is the minimum size that will hold the component)
BorderLayout	Varies depending on location: • North and South components' height preference respected, width ignored • East and West components' width preference respected, height ignored • Center component's height and width preferences ignored
GridLayout	Width and height preferences ignored
CardLayout	Width and height preferences ignored
GridBagLayout	Varies depending on component's GridBagConstraints

As you can see, components layed out using FlowLayout are not resized at all. Using BorderLayout, certain components are resized in certain directions.

8.7. Sizing Components using Preferred Size

When using either GridLayout or BorderLayout, components are automatically resized to fill their container. A component's `preferredSize()` method can be overridden (by subclassing the component) to control sizing.

For example, the Button class could be subclassed to specify a preferred size:

```
class SizedButton extends Button {
    public SizedButton(String label) {
        super.setLabel(label);
    }
    public preferredSize() {
        return new Dimension(150,20);
    }
}
```

Subclassed buttons could then be declared, created, and added to the interface as follows:

```
SizedButton newButton;

newButton = new SizedButton("Sized");
add("East",newButton);
```

A `minimumSize()` method is also defined, but is ignored most of the time.

8.8. Layout Strategies

Laying out the user interface can be one of the most challenging aspects of Java programming. Following are a couple of tips for creating useful interfaces:

1. Take advantage of each layout manager's strengths:

    - FlowLayout is good for laying out components you do not want resized
    - BorderLayout is good when you want one component to expand and fill the remainder of a container (place this component in the "Center")
    - GridLayout is good when you want equally-sized components
    - CardLayout is good when you have several "screenfuls" of components, but only one is visible at a time

2. Group components into panels, each using their own layout manager

3. Subclass components to override their preferred size when necessary

Note: Most layout situations can be handled using the four layout managers described here. A fifth layout manager, GridBagLayout, is also provided by the AWT, but is much more complex.

9. Additional Components

Frames and dialog boxes are subclasses of Window, which is a subclass of Container. Therefore, you can add components to a frame or dialog box just like you would to an applet.

9.1. Frames

Frames are used to create free-standing windows, either to add an additional window to an applet, or to create a graphical user interface for a standalone application:

Constructors	`Frame()` `Frame(String title)`

Public Methods	`setTitle(String t)`
	`setLayout(LayoutManager lm)`
	`add(Component c)`
	`resize(int w,int h)`
	`move(int x,int y)`
	`show()`
	`hide()`
	`dispose()`

Events	Window events generated

In order to distinguish applet frames from other windows, a warning message is displayed. This message may appear as follows:

- "Unsigned Java Applet Window"
- "Warning: Applet Window"

Frame objects themselves can be created. In order to respond to window events, however, a subclass of Frame is created.

```
import java.awt.*;

class FrameSubclass extends Frame {
    // Assume we are in an applet
    boolean inAnApplet = true;

    // Constructor method
    public FrameSubclass() {
        // Set title, size, and move into position
        setTitle("My Frame");
        resize(300,200);
        move(250,350);

        // Add components
        add("North",new Label("This is my Frame"));
        add("Center",new TextArea(4,20));
    }

    // Handle window events
    public boolean handleEvent(Event e) {
        if (e.id == Event.WINDOW_DESTROY) {
            hide();
            dispose();
            if (!inAnApplet)
                System.exit(0);
            return true;
        }
        return super.handleEvent(e);
    }
}
```

Example 7 FrameSubclass.java

The following steps are used to create frame:

- instantiate a new Frame subclass object
- set the frame's layout (the default is BorderLayout)
- add components to the frame
- size the frame and move it into position
- make the frame visible

The Frame subclass can then be used to create a window in a standalone application or to add an additional window to an applet.

```
class FrameApplication {
    public static void main(String args[]) {
        // Create and show frame
        FrameSubclass myFrame = new FrameSubclass();
        myFrame.show();

        // Set inAnApplet variable
        myFrame.inAnApplet = false;
    }
}
```

Example 8 FrameApplication.java

Note: To exit properly from an application, we use the `System.exit()` method. This need not be called in an applet, so we keep track of the program type using the `inAnApplet` variable.

The following creates a frame in an applet:

```
import java.applet.Applet;
import java.awt.*;

public class FrameApplet extends Applet {
    public void init() {
        FrameSubclass myFrame = new FrameSubclass();
        myFrame.show();
    }
}
```

Example 9 FrameApplet.java

The application and applet are similar and can be combined:

```
import java.applet.Applet;
import java.awt.*;

public class Combination extends Applet {
    public void init() {
        createFrame(true);
    }

    public static void main(String args[]) {
        createFrame(false);
    }

   public static void createFrame(boolean inAnApplet) {
        FrameSubclass myFrame = new FrameSubclass();
        myFrame.inAnApplet = inAnApplet;
        myFrame.show();
    }
}
```

Example 10 Combination.java

9.2. Dialog Boxes

Dialog boxes are used to gather simple input from the user or to display status, alert, or warning messages. Dialog boxes are similar to frames, but cannot contain a menu bar. Dialog boxes may be modal—all other use of the application is blocked until the dialog box is dealt with.

Constructors	`Dialog(Frame f,`
	`  boolean isModal)`
	`Dialog(Frame f,`
	`  boolean isModal,`
	`  String title)`

Public Methods	`setResizable(boolean b)`
	`setLayout(LayoutManager lm)`
	`add(Component c)`
	`resize(int w,int h)`
	`move(int x,int y)`
	`show()`

Events	none

Modal dialog boxes cannot be moved and do not allow users to switch to other windows in the same program.

9.3. File Dialogs

FileDialog is a subclass of Dialog that brings up the native file chooser, allowing a filename to be selected. Most useful in applications due to file access restrictions in applets:

```
import java.awt.*;

public class fd {
  static Frame f = new Frame("my frame");

  public static void main(String[] a) {
    FileDialog fd =
        new FileDialog(f, "my filedialog");
    fd.show();
    f.add(fd)
    f.show();
  }
}
```

Example 11 FileDialog.java

String to retrieve the filename:

```
String s = fd.getFile();
```

File dialogs look quite different on different operating systems.

Note: FileDialogs were too buggy to use in 1.02 but have improved dramatically in 1.1.

9.4. Menus

MenuBars are components that can be added to the top edge of a Frame. (MenuBars cannot be added to the applet window.) A MenuBar can consist of many individual pull-down menus, which themselves are made up of menu items.

```
MenuBar mb;
Menu fileMenu;

mb = new MenuBar();
fileMenu = new Menu();
fileMenu.add("New");
fileMenu.add("Open...");
fileMenu.add("Save");
fileMenu.add("Save As...");
fileMenu.add("-");
fileMenu.add("Quit");

mb.add(fileMenu);
setMenuBar(mb);
```

Menus can be nested. For example, a submenu could be created and added to the fileMenu above. Checkbox menu items are also available.

10. Object Oriented Programming (Part Two)

10.1. Data Hiding

Accessibility of variables and methods can be restricted in order to hide unimportant implementation details of a class. Variables are typically made "private" and are accessible only through pairs of public "set" and "get" methods.

This "*encapsulation*" of a class's variables has two benefits:

- Modularity — Algorithms can be changed and improved without changing a class's usage or appearance to the outside world
- Robustness — Data can be validated in the "set" method and results properly formatted in the "get" method

Variable and method access specifiers:

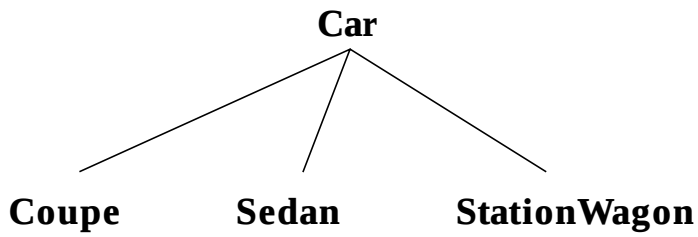
- `public` — Accessible anywhere the object can be seen
- `private` — Accessible only by the object itself or by other objects of the same class
- `protected` — Accessible only by objects in the same package (directory) as the class and by objects in a subclass of the class
- unspecified — Accessible to anything in the same (or "friendly")package, but not to any subclasses that are in different packages

By default, classes can only be used by other classes in the same package. Specifying a class as `public` allows the class to be used anywhere.

10.2. Inheritance

Classes can be created from scratch or by extending existing classes, inheriting their variables and methods. Subclasses can add instance variables and methods of their own. They can also override (redefine) the superclass's variables and methods.

For example, the Car class defined previously can be extended to define specific types of cars:



A coupe could thus be defined as follows:

```
class Coupe extends Car {
    float topSpeed;
    int numDoors = 2;
}
```

Subclasses can access public, protected, and "friendly" members (variables and methods) of its superclass, but not their private members.

Notes: Every object of a subclass is also a member of its superclass. Multiple levels of inheritance can occur.

Classes can be declared `final`, meaning they cannot be subclassed.

10.2.1. Subclass Constructors

When creating a new subclass object, a no argument constructor of its superclass (either the default no-argument constructor or a programmer defined no-argument constructor) is called first, followed by the subclass constructor.

One exception to this calling of the superclass's no-argument constructor is if the superclass constructor is referenced in the first line of the subclass constructor:

```
public Coupe(String make,
             String model,
             float topSpeed) {
    super(make,model);
    this.topSpeed = topSpeed;
}
```

In this case the superclass constructor specified using the `super` keyword is called first. This allows us to initialize superclass variables as well as subclass variables.

10.2.2. Method Overriding

A subclass can "override" a superclass method to provide a specialized implementation. For example, Car's `toString()` method could be overridden in the Coupe subclass:

```
public String toString() {
  return numDoors + " door " +
         make + " " +
         model + " coupe " +
        "(top speed " + topSpeed + " mph)";
}
```

10.2.3. The Object Class

Actually, classes are never really created from scratch. Every class inherits from `java.lang.Object`, the root of the class hierarchy. The Object class defines the basic state and behavior that all object must have.

Methods that the Object class provides include:

- `toString()` Returns the name of the class (typically overridden in programmer-defined classes)
- `clone()` Creates an exact clone of an object in its present state (Netscape applets cannot use clone for security reasons)
- `equals()` Tests whether two objects are equal to each other in all respects (not the same as ==, which tests if two objects are the same object)
- `getClass()` Returns a class descriptor of an object that you use to test the type of a particular object at runtime

10.3. Packages

Groups of related classes can be organized into a class library, or in Java terminology, a "*package*." This makes objects both easier to find and to use, and avoids naming conflicts. (Packages can also contain interface definitions.)

To use the classes and interfaces defined in one package from within another package, you need to import the package. For example:

```
import java.applet.Applet;
import java.awt.*;
```

The first line imports a specific class from the java.applet package. The second line import all classes from java.awt. The java.lang package is required by almost every Java program, and is automatically imported for you.

A package can be defined as follows:

```
package addresses;

public class Person {...}

public class AddressBook {...}
```

The "`.class`" files corresponding to the class definitions in the package must be placed in a directory with same name as the package (in this case "`addresses`") and must be included somewhere in your CLASSPATH definition.

The default package (a package with no name) is always imported for you.

11. Input and Output

Java's I/O system is very flexible, allowing you to read from and write to the console, files on the local system, and files on a Web server.

11.1. Streams

In Java, all input and output requires the use of a stream object, three or which are created automatically for console I/O:

- `System.in`
- `System.out`
- `System.err`

These streams (along with FileInputStreams and FileOuput-Streams used with file I/O) allow you to read or write data byte by byte. To make I/O easier, one of the following filters can be "chained" to one of the more basic streams:

- DataInputStream reads primitive data types
- DataOutputStream writes primitive data types
- PrintStream writes ASCII text

Other types of streams perform buffered I/O, line-numbered I/O, and I/O between pipes (allowing threads of execution to communicate).

Notes: I/O frequently cause IOExceptions to be thrown—try-catch blocks must be used (except when writing to the console).

11.2. Console I/O

Lines of data can be read from the console by first chaining a DataInputStream object to the System.in stream and then using the DataInputStream object's `readLine()` method:

```
try {
    DataInputStream myInput =
        new DataInputStream(System.in);

    System.out.print("Enter your name: ");
    String name = myInput.readln();
    System.out.println("Hello " + name);
}
catch (IOException e) {
   System.err.println("IOException: " + e);
}
```

Example 12 ConsoleIO.java

This example also shows the use of System.out's `print()` and `println()` methods (`println()` automatically appends a newline character).

Note: PrintStreams (such System.out and System.err) are buffered. Data remaining in the buffer when a program crashes is not written, possibly leading you to think the program crashed at some other point. The buffer can be flushed using:

```
System.out.flush()
```

11.3. File I/O

Files are opened by creating FileInputStream and FileOutputStream objects. DataInputStream and DataOutputStream filters can then be used to perform I/O of primitive data types.

11.3.1. Reading From a Local File

The following code snippet demonstrates how to read data line by line from a file on the local file system:

```
String line;
try {
  DataInputStream input =
    new DataInputStream(
      new FileInputStream("quote.txt"));

  while ((line = input.readLine()) != null)
    System.out.println(line);

  input.close();
}
catch (IOException e) {
  System.err.println("IOException: " + e);
}
```

Example 13 ReadFile.java

11.3.2. Writing To a Local File

The following example writes lines to a local file:

```
try {
  PrintStream output =
    new DataOutputStream(
      new FileOutputStream("out.dat"));

  output.println("Help me...");

  output.close();
}
catch (IOException e) {
  System.err.println("IOException: " + e);
}
```

Example 14 WriteFile.java

In this example, if "out.dat" already exists, it will be overwritten. There is no means of writing at the end of a sequential file.

When using chained stream objects, the outermost object should be used to close the file.

```
output.close();
```

Note: Random access to files is also possible, enabling instant access to individual records. Random access files are usually created using fixed length records.

11.3.3. Local File Information

The File class can be used to gather information about a local file. First we create a new File object for the file in which we are interested:

```
public File(pathname);
public File(directoryString,filename);
public File(directoryFile,filename);
```

Once we have a file object, we can invoke its various methods. For example:

```
File theFile = new File("quote.txt");

if (theFile.exists()) {
  if (theFile.isFile()) {
    lastModified = theFile.lastModified();
    length = theFile.length();
  }

  else if (theFile.isDirectory()) {
    String dir[] = theFile.list();
  }
  else
    System.out.println("Not regular file");

else {
  System.out.println("File doesn't exist");
}
```

Example 15 FileStatus.java

11.4. Web Server I/O

Java allows applets and applications to display Web pages in the browser, and also to read and write files on a Web server.

11.4.1. Displaying Web Pages

The following example displays a Web page:

```
try {
  URL myPage = new URL(urlString);
}
catch (MalformedURLException e) {
  System.err.println(e);
}
getAppletContext().showDocument(myPage);
```

Example 16 DisplayPage.java

The `getAppletContext()` method returns an object representing the applet's environment, ie. browser. The URL constructor must be called in a try-catch block, or a throws clause must be specified on the method definition.

11.4.2. Reading a File on a Server

The following example demonstrates how a Web server file can be read:

```
String line;
try {
  URL url = new URL(someURLstring);
  DataInputStream input =
    new DataInputStream(url.openStream());

  while ((line = input.readln()) != null)
    System.out.println(line);

  input.close();
}
catch (MalformedURLException e) {
  System.err.println("MalformedURL: " + e);
}
catch (IOException e) {
  System.err.println("IOException: " + e);
}
```

Example 17 ReadServerFile.java

In this example, we create a URL object and then use its `openStream()` method to open a connection to the file.

11.4.3. Writing a File on a Server

Writing to a URL is not directly supported. It can be accomplished indirectly, however, by "posting" data to a CGI script running on the server. This works as follows:

1. Create a URL.
2. Open a connection to the URL.
3. Get an output stream from the connection. This output stream is connected to the standard input stream of the CGI script on the server.
4. Write to the output stream.
5. Close the output stream.

11.5. Common Escape Sequences

Escape sequences can be used to output special characters, including:

\n	newline
\t	tab
\r	return (to beginning of current line)
\’	apostrophe
\"	quotation mark

12. Multithreading

Threads (an abbreviation for “threads of control”) are the way to get more than one thing to happen at once in a program. There are a variety of reasons why this would be desirable. For example, you might want your program to be able to respond to its GUI while working on some other task or design a client / server program where the program spawns new threads in response to separate client requests. Threads are also closely related to parallel processing which may be implemented on certain multiprocessor machines.

There are two ways to obtain a new thread, by extending java.lang.Thread or by implementing the Runnable interface.

12.1. Extending the Thread Class

Extend class java.lang.Thread and override “`run()`”:

```
class MyThreadClass extends Thread {
   public void run() {
      ...
   }
}
```

Overriding the `run()` method is necessary. `run()` is not called directly; it is run for you and behaves like a call to `main()`. Threads do not start running on creation:

```
MyThreadClass x = new MyThreadClass();
```

You start the thread by calling the `start()` method as in `x.start()` or in one step, like this:

```
new MyThreadClass().start();
```

which in turn will execute the `run()` method of your threaded class. The Thread class includes methods to `stop()`, `sleep()`, `suspend()`, `resume()`, `setPriority()`, `getPriority()`, etc.

12.2. Implementing the Runnable Interface

You can not extend the Thread class in an applet since applets already extend class Applet by definition (you are not allowed to subclass more than one class). You can, however, implement the Runnable interface which looks similar to the extension of the Thread class:

```
class MyThreadClass implements Runnable {
   public void run() {
      ...
   }
}
```

Unfortunately the methods in the Thread class are not available to a non subclass of Thread. In order to kludge around this, you must create a new object from the Thread class and pass it your Runnable class:

```
Thread th = new Thread(new MyThreadClass());
```

Note: you still cannot call Thread methods from with the Runnable interface implementation.

A simple example of a threaded application:

```
public class drinks {
  public static void main (String[] a) {
    new Beer().start();
    new Wine().start();
  }
}

class Beer extends Thread {
  public void run() {
    while (true) {
      System.out.println(
         “I’ll have a beer”);
      yield();
    }
  }
}

class Wine extends Thread{
  public void run() {
    while (true) {
      System.out.println(
         “I’ll have a Wine”);
      yield();
    }
  }
}
```

Example 18 drinks.java

This above program will repeat until you press control-C. Note the call to `yield()` which gives up control of the process to other threads to insure that one thread does not hog all the CPU. (This would not have been necessary if time-slicing was a standard part of Java.)

12.3. Writing Thread Safe Code

In the “beer or wine” example, we used totally unrelated threads that did not rely on each other, nor use any common objects. When threads operate on or read the same objects or memory space, there could be a problem with race conditions. Because there is no way to be sure a thread will be allocated resources in a FIFO (first in, first out) manner, there is no automatic way to guarantee exactly when a thread will begin and at what speed it will execute.

Consider the following example of a race condition between ten threads that all check and raise the pressure it if it’s not above a critical value:

```
public class pressure {
  static int pressureGauge = 0;
  static final int safetyLimit = 20;

  public static void main(String args[]) {
    CheckPressure p1[] = new CheckPressure[10];

    for (int i=0; i<10; i++) {
      p1[i] = new CheckPressure();
      p1[i].start();
    }
    try {
      for (int i=0;i<10;i++) p1[i].join();
    }
    catch (Exception e) {}

    System.out.println("guage reads" + pressureGauge +
                       ", safelimit is" + safetyLimit);
  }
}

class CheckPressure extends Thread {
  public void run () {
    RaisePressure();
  }

  void RaisePressure() {
    if (pressure.pressureGauge <
        pressure.safetyLimit-15) {
      try {
        sleep(100);
      }
      catch (InterruptedException e) {}
      pressure.pressureGauge += 15;
    }
  }
}
```

Example 19 pressure.java

The `sleep()` method is inserted here to insure that there will be race conditions in the code. It stops the code for a desired number of milliseconds. The `join()` method forces the code to wait till the thread is finished before continuing. The output of this code is: “gauge reads150, safelimit is20”. Clearly we did not get the desired effects due to the competition between threads to write to the same pressure variable.

12.4. Controlling Thread Execution

Writing thread-safe code so that the threads stay out of each other’s way is known as "*mutual exclusion*." To make a portion of code mutually exclusive you use the `synchronized` keyword. There are essentially two ways to use synchronization.

12.4.1. Synchronization at the Method Level

Mutual exclusion of a method over an entire class can be achieved by adding “static synchronized” keywords and would provide the same.

```
static synchronized void RaisePressure() {
    ...
}
```

This would mean that only one instance of the class could be in the method RaisePressure at any particular time.

12.4.2. Synchronization at the Block Level

Enclosing a block of code in parentheses preceded by `synchronized(Object Obj)` allows you to synchronized a portion of a method:

```
static Object Obj = new Object;

void RaisePressure() {
  synchronized(Obj) {
    if (pressure.pressureGauge <
        pressure.safetyLimit-15) {
      try {
        sleep(100);
      }
      catch (Exception e) {}

      pressure.pressureGauge += 15;
    }
  }
}
```

We must provide an object which is available to all threads and therefore static to act as a key. A thread must grab the key to enter a code block and gives up when it leaves. We could have used an existing object in our example if we had one.

12.5. Communicating Between Threads

Sometimes it is not just necessary to keep other threads out of blocks of code at the same time but to have points in the code where you need to give up / take control to / from another thread.

- notify() Release the lock and suspend.
  Wakes up a single thread that is waiting on this object's monitor. A thread waits on an object's monitor by calling one of the wait methods.

- notifyall() Wakes up all threads that are waiting on this object's monitor. A thread waits on an object's monitor by calling one of the wait methods.

- wait() Gives up lock and suspends until notified.
  Waits to be notified by another thread of a change in this object. The current thread must own this object's monitor. The thread releases ownership of this monitor and waits until either of the following two conditions has occurred:

  1. Another thread notifies threads waiting on this object's monitor to wake up either through a call to the notify method or the notifyAll method.
  2. The timeout period, specified by the timeout argument in milliseconds, has elapsed.

Only one thread at a time can own an object's monitor. The notify(), wait() and notifyall() methods should only be called by a thread that is the owner of this object's monitor. A thread becomes the owner of the object's monitor in one of three ways:

1. By executing a synchronized instance method of that object.
2. By executing the body of a synchronized statement that synchronizes on the object.
3. For objects of type Class, by executing a synchronized static method of that class.

- interrupt() Waking up a sleeping / waiting thread. The interrupt method throws an exception that must be caught by the sleeping thread using a try / catch block.

```
try {
  sleep(some time);
}
catch (InterruptedException e) {...}
```

13. Multimedia

Java has built in capabilities to load and display images, run animations, and play audio.

13.1. Loading Images

Two image formats are currently supported—GIF and JPEG. Images can be downloaded from a URL or loaded from a local file (applications only).

Two identical `getImage()` methods are defined—one in the `java.applet.Applet` package (for use in applets), the other in `java.awt.Toolkit`.

Applets can download images from a Web server using either:

```
Image myImage = getImage(URL);
Image myImage = getImage(URL,filename);
```

Applications can load images from a local file or from a Web server by first getting the toolkit associated with the current graphics object and then using its `getImage()` method:

```
Image myImage =
    Toolkit.getDefaultToolkit().
        getImage(filenameorURL);
```

Note: A separate thread of execution is launched to download images, enabling execution to continue.

Image size can be obtained using `getWidth()` and `getHeight()`, both of which return -1 until the image is fully loaded.

13.2. Displaying Images

Once an image has been loaded, it can be displayed in its original size or automatically scaled:

```
drawImage(myImage,
          x,y,
          ImageObserver observer);

drawImage(myImage,
          x,y,
          width,height,
          ImageObserver observer);
```

Both of these methods take an ImageObserver argument, which is normally the component on which the image is displayed. This enables the component to be notified of progress during the image loading phase. Most components can simply specify `this`. For example:

```
drawImage(myImage,0,0,this);
```

By obtaining its width and height, an image can be scaled proportionally:

```
drawImage(myImage,
          0,0,
          myImage.getWidth()*2,
          myImage.getHeight()*2,
          this);
```

Note: The `drawImage()` method also runs asynchronously.

13.3. Playing Audio Clips

Sounds can be played in an applet a single time (using Applet's `play()` method or continuously (using the AudioClip interface).

Either of the following methods can be used to download a sound from a Web server and play it one time:

```
play(URL);
play(URL,filename);
```

To play a sound continuously, you first get a reference to an AudioClip object:

```
AudioClip au = getAudioClip(URL);
AudioClip au = getAudioClip(URL,filename);
```

and then use its loop method:

```
au.loop();
```

To stop the sound, use:

```
au.stop();
```

Currently, only 8-bit, 8000 Hz, one-channel Sun ".au" files are supported.

References

Many online and hardcopy references were used in creating this class. Among the best of these are the following:

- "Just Java"

  by Peter van der Linden
  SunSoft Press (Prentice Hall)
  ISBN: 0-13-272303-4

- "Graphic Java"

  by David M. Geary and Alan L McClellan
  SunSoft Press (Prentice Hall)
  ISBN: 0-13-565847-0

- The Chen and Lee book (I forget the name)

- Sun Microsystem's Java Web page at:

  `http://java.sun.com/`

- The Gamelan Java Site:

  `http://www.gamelan.com/`

- Java newsgroups

Summary

The Java Development Kit is available free of charge from Sun Microsystems (http://java.sun.com/). Versions are available for Windows 95/NT, Solaris, and other operating systems.

There are several steps to the Java learning process:

1. becoming familiar with object-oriented principles
2. learning the Java language itself
3. learning to use classes and methods in Java class libraries

This class was designed as an introduction to Java. Due to its popularity and the pace of innovations occurring in the computer industry today, Java has evolved significantly in many areas. But according to the "80/20" rule, while what we have covered in this class represents only 20% of the Java language, it is what you will be using 80% of the time.

In this class we developed Java programs "from scratch." There are several Integrated Development Environments (IDEs) also available.

Hopefully these notes will help you efficiently get started on the road to becoming a Java programmer.